CS 6240 SECTION - 1

PROJECT: CLASSIFICATION OF RED WING BIRD

SANDEEP LAGISETTY & SUJITH MAHENDRAKAR

Goal: The goal is to predict the presence or absence of the Red-winged Blackbird (Agelaius phoeniceus) in each birding session as accurately as possible. This bird is coded in the 27th column of the data sets; its name is in the header for that column.

Dataset: This dataset contains count data for bird species observed by novice and experienced bird observers (a.k.a. birders). The data was submitted by volunteers to the eBird Citizen Science Project run by the Cornell Lab of Ornithology. A record in this dataset corresponds to a checklist that a birder uses to mark the number of birds of each species detected; one checklist is submitted per sampling event (i.e. birding session). Each checklist submitted from locations in the Western Hemisphere is additionally annotated with predictor variables (called covariates below) that are derived from the location of the sampling event.

Link to Dataset: https://drive.google.com/drive/folders/0BxLUDit8x6n1VXRTOC1TdEpCNEE

Reference document describes the data in its original file structure, where the different variables (called covariates in the document) of an observation record were distributed over three files:

1. Checklist - 16 columns describing the sampling event (date, time, latitude, longitude, duration, distance traveled, protocol, observer id, etc.), plus variables for all the species in the data set. There is one column per species, with each column listing the number of birds observed.

2. Core covariates - A priori are most important for most of the species.

3. Extended covariates - These include extensive statistics about habitat configuration, fine-grained climate measurements, and observer expertise.

Note: Information across all the individual files are combine to let the data appear in its intended un-partitioned beauty. In particular, all covariates appear together in a single line for each record.

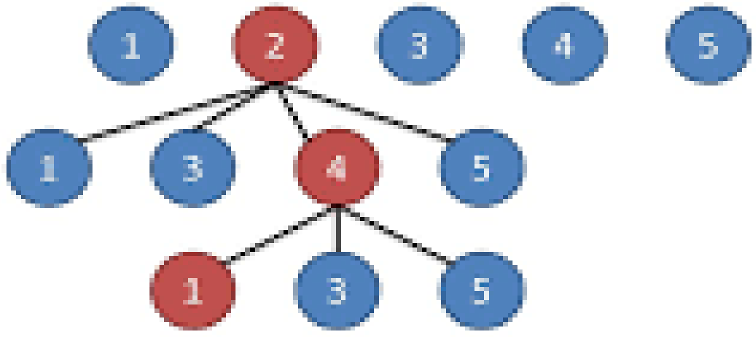
Tools & Languages used: We have used Hadoop Framework, Java, Weka, Gradle, AWS S3 & EMR to develop the project and test the project.

Weka is a collection of machine learning algorithms developed for data mining tasks developed by Machine Learning group at University of Waikato, New Zealand. It is open source software issued under the GNU General Public License.

DESIGN: For Training the model there's a mapreduce job. In mapper each line from the csv file is read and emitted out as value with a random number between 1-10 as key. Then all the rows with same key reach the same reducer. In the reducer we split the csv and load into Weka's Instances method. The next step is to preprocess the data.

Preprocessing: Since the data is uncleaned, not preprocessed we need to preprocess it. There are also ~1800 attributes out of which there are 700 bird species which doesn't contribute for the prediction red wing bird. Similarly, SAMPLE_ID has no importance in predicting the presence of red wing bird. As we know about curse of dimensionality, we need to select the attributes from the ~1000 list of attributes.

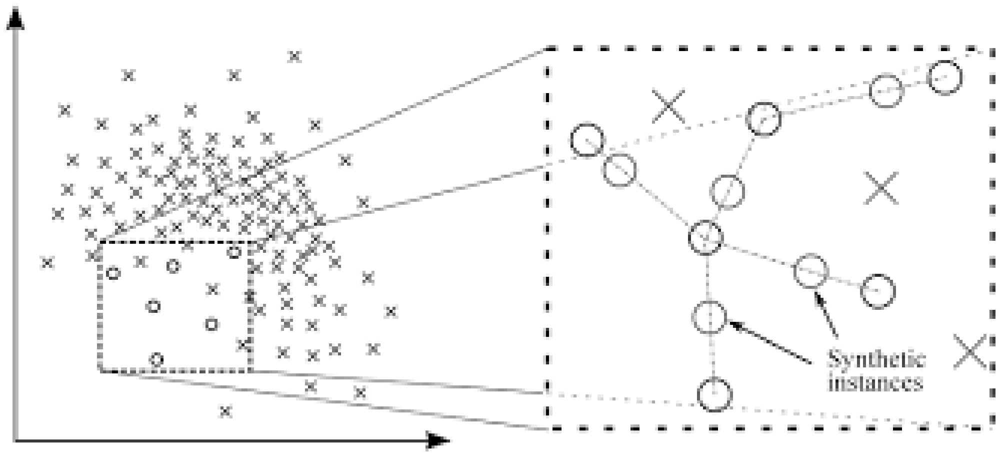
<u>Wrapper Method</u>: To filter out the attributes we have used Weka's attribute selection option which uses wrapper method. It initially starts with no features, greedily include the most relevant feature.



As mentioned in the above diagram, it starts with a 2 feature, then selects 4, then 1 and goes on. The same applies to our dataset. The next step is to drop the attributes from the weka's instances and includes only the ones which the wrapper method suggested. We have used Weka's drop attribute to remove the attribute.

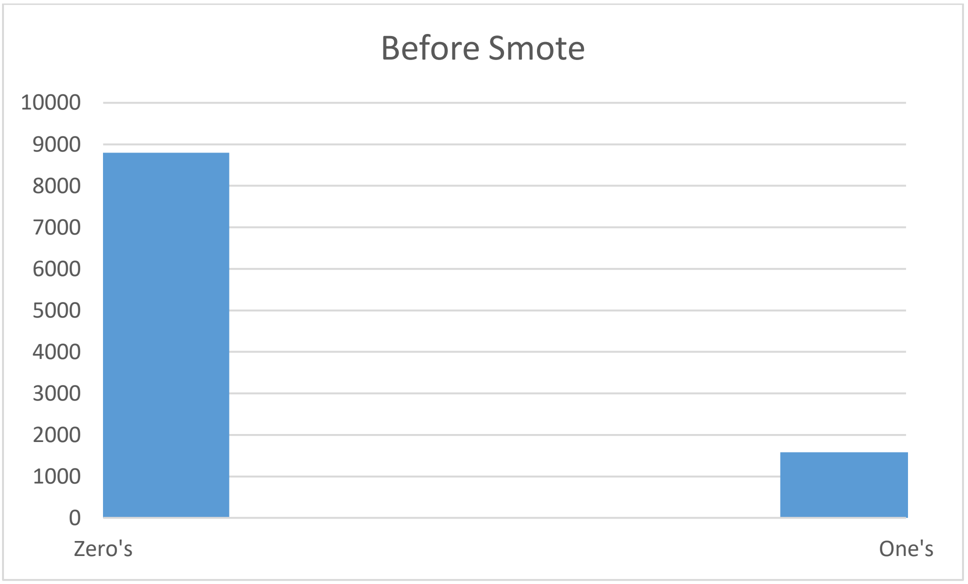
The next step is missing value replacement. Since the data is in its natural form, there are lots of missing values and Noise. Therefore, we need to handle it. The given data set has noise, like string values for latitude and longitude and also since the data is set by observers there might be some attributes which might not be observed, therefore there are "?" in some on the instances. We use Weka's missing value imputation method to remove noise and missing values. Since all the attributes are continuous numeric values, we replace the missing value by the median of that attribute.

We know that all the standard learner algorithms are often biased towards the majority class. This is because these classifiers attempt to reduce the global error rate, not taking into account the data distribution. As a result, examples from the majority classes are well-classified whereas examples from the minority class tend to be misclassified. To solve this problem, we have used Weka's smote method. Smote method randomly generates synthetic samples along the lines joining the minority sample and its r-1 selected neighbor's.

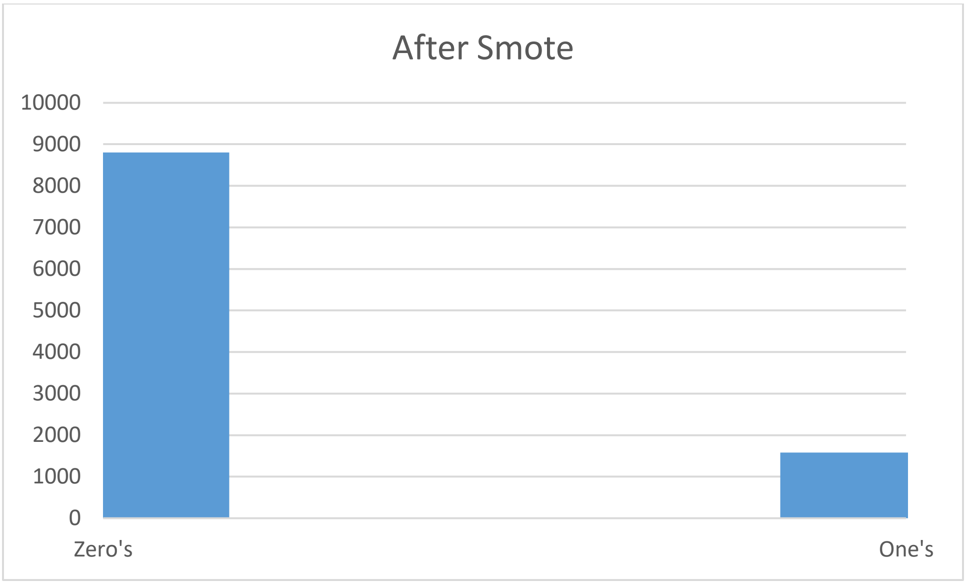


The next process in preprocessing is to convert numeric attributes into nominal attributes. The reason behind this is that most of Weka's algorithms supports nominal attributes. We use weka's numericaltonominal filter to do this.

Example: Consider the histogram for zero's and one's at a reducer. The total count of zero's are 8798 and one's are 1202.



The below histogram is after applying smote. Though the number of zero's remain same, there are increase in one's.



Building Classifier: We have trained J48 (C4.5 Decision Tree), RepTree and Random Forest algorithms on the dataset and built classifiers.

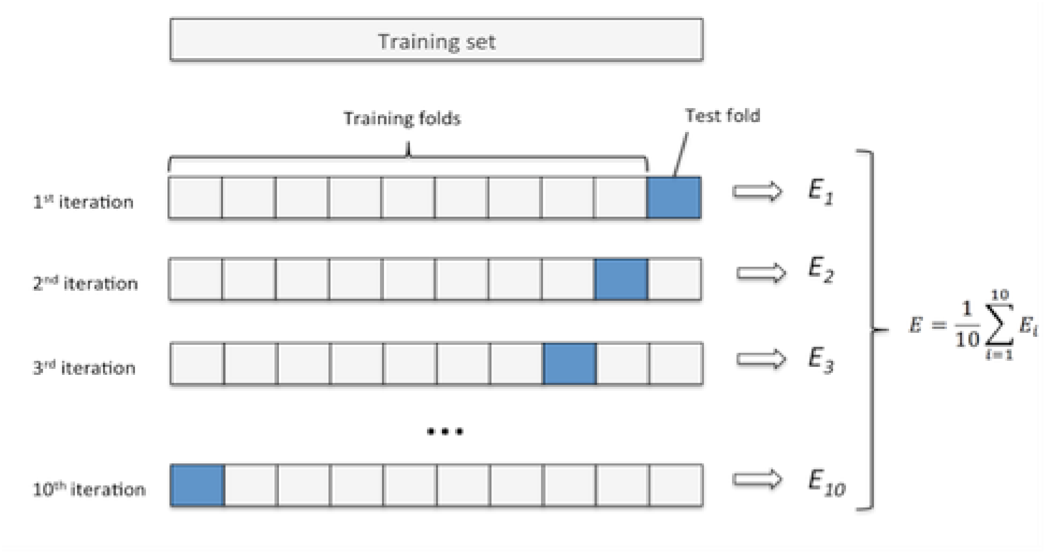
J48: **J48** is an extension of ID3. The additional features of **J48** are accounting for missing values, decision trees pruning, continuous attribute value ranges, derivation of rules, etc. In the WEKA data mining tool, **J48** is an open source Java implementation of the C4.5 **algorithm**.

RepTree: Fast decision tree learner. Builds a decision/regression tree using information gain/variance and prunes it using reduced-error pruning (with backfitting). Only sorts values for numeric attributes once. Missing values are dealt with by splitting the corresponding instances into pieces (i.e. as in C4.5).

RandomForest: **Random forests** or random decision forests are an ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of overfittingto their training set.

Every classification algorithm has some parameters and they needed to tuned. Therefore, we have used cross validation to tune all the algorithms.

Cross Validation: We divide the data into K folds. We train the algorithm on (k-1) folds and then test it on the kth fold. We measure it on the average performance.



For J48 parameters like confidenceFactor are tuned. For RandomForest max depth numoftrees are tuned and for RepTrees maxDepth and minVarianceProp are tuned.

After training these algorithms we serialize the trained objects into HDFS and deserialized when used on test instances.

We write another mapreduce job for predicting the test instances. Mapper and Reducer are almost as the previous one, except the trained classifier is deserialized and is used on test instance to classify.

PSEUDO CODE:

```
Class TrainingMapper{
        Map(Offset B, Line L)
        rand = RandomGeneratorInt(0,10):
        Context.write(rand , line l);
}
```

<u>Design</u>: The reason we have used random generator is to partition the huge dataset and also distribute the load among multiple reducers. We inititally thought of reducing partitioning by year but the range of the year in dataset is 2008-2016. Therefore, this cannot be scaled up. The other way is to include (month, year) to partition but that becomes a tedious process. Therefore, we have used a random integer as key to partition the data. This way it can be easily scaled up by increasing the range of random integer generator. Since numbers obtained from random generated are normally distributed, each reducer gets equal amount of data.

Suppose there are 100000 rows in csv file, each reducer gets ~1/10th of total data. But this can be scaled up by using more range in random integer generator. Consider it is increased to 20. Therefore, if we have 20 machines, each reducer would get ~1/20th of total data.

```
Class TrainReducer(Int key, [Text1, Text 2,…]){
    Weka.Instances data;
    For (each : values){
        Instance = each.string();
    data.addd(instance);
    }
    Weka.AttributeFilter filter;
    Options <- Set_Parameters;
    Filter.setOptions(options);
    data = Filter(filter, data);

    Weka.MissingValueImputation filter;
    Options <- Set_Parameters;
    Filter.setOptions(options);
    data = Filter(filter, data);

    Weka.Smote filter;
    Options <- Set_Parameters;
    Filter.setOptions(options);
    data = Filter(filter, data);
    Weka.NumericalToNominal filter;
    Options <- Set_Parameters;
    Filter.setOptions(options);
    data = Filter(filter, data);
```

```
        Crossvalidation (each_algorithm): {
            For 10 times:
                divide train data into 10 parts
                 for i = 1 to 10
                      train each_algorithm using 9 parts
                       compute accuracy using 1 part
                 compute average accuracy of the 10 runs
        each_algorithm <- highest_avg_accuracy
        serialize_object(each_algorithm)
        }
        If(key = 1 || 2 || 3}{
             CrossValidation(j48)
        } else if (key = 4 || 5 || 6) {
             CrossValidation(repTree)
        }else
             CrossValidation(randomForest)
}
```

Design: In reducer we first build the instances and then filter the attributes using weka's wrapper method -> then we remove missing values and noise using weka's missing value imputation method -> then to remove the imbalance we use weka's smote method -> then convert the numerical attributes to nominal attributes. We then do cross validation to find the best tuned parameters for specific algorithm and then serialize that classifier object to HDFS.

```
Class ClassifyMapper{
        Map(Offset B, Line L)
        rand = RandomGeneratorInt(0,10):
        Context.write(rand , line l);
}
```

Design: ClassifyMapper is similar to TrainMapper where we use a random integer generator to partition and load balance data. The reason we have used random generator is to partition the huge dataset and also distribute the load among multiple reducers. We inititally thought of reducing partitioning by year but the range of the year in dataset is 2008-2016. Therefore, this cannot be scaled up. The other way is to include (month, year) to partition but that becomes a tedious process. Therefore, we have used a random integer as key to partition the data. This way it can be easily scaled up by increasing the range of random integer generator. Since numbers obtained from random generated are normally distributed, each reducer gets equal amount of data.

```
Class TrainReducer(Int key, [Text1, Text 2,…]){
    Weka.Instances data;
    For (each : values){
        Instance = each.string();
    data.addd(instance);
    }
    Weka.AttributeFilter filter;
    Options <- Set_Parameters;
    Filter.setOptions(options);
    data = Filter(filter, data);

    Weka.NumericalToNominal filter;
    Options <- Set_Parameters;
    Filter.setOptions(options);
    data = Filter(filter, data);

    Classifiers algorithms = [j48-1,j48-2,j48-3,…,repTree-3,…randomforest-4]
    Foreach.algorithms(each -> deserialize)
    // Ensemble – Voting
    For (i =0 to i < data.numinstances){
        Int zero <- 0, int one <- 0
        If (Foreach.algorithms(classifyInstance(i) == 0)
            Zero ++
        Else
            One++
        If (zero > one)
            Emit(0,null)
        Else
            Emit(1,null)
    }
}
```

Design: Preprocessing of test instances are done similar to train preprocessing except that smote and missing value filters are not applied. Next all the 10 different models are deserialized and each test is classified by 10 different models and a majority vote is taken among them. If there are more zero's then zero is written to file else 1 is written to file.

The reason we do ensemble learning is that when we combine multiple independent decisions, each of which is at least more accurate than random guessing, random errors cancel each other out and correct decisions are reinforced.

Quality:

I have trained the algorithms on entire data set except the last 10000 rows. I considered them as test data and when ran against the that set, I got the following results:

```
=== Confusion Matrix Random Forest-1 ===

   a    b   <-- classified as

 7418  764 |    a = 0

 1514  304 |    b = 1
```

Accuracy: 77.22%

```
=== Confusion Matrix  RepTree-1===

    a    b   <-- classified as

 7390  792 |    a = 0

 1596  222 |    b = 1
```

Accuarcy: 76.12

```
=== Confusion Matrix for J-48-1===

    a    b   <-- classified as

 6808 1374 |    a = 0

 1435  383 |    b = 1
```

Accuracy: 71.91%

Ensemble Learning Accuracy for 10 models = 79.8%